**Lab06**

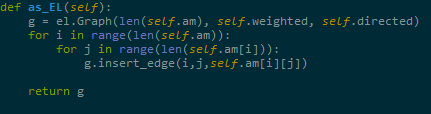
In lab 06 we are asked to implement different graph functions. We are asked to implement all the same functions using matrix representation and edge list representation. We are given three files that all represent different representations of graphs. In each of the files, the first step is to implement the functions, insert\_edge, delete\_edge, and display. Each of the functions are slightly different but can be implemented in a similar manner.

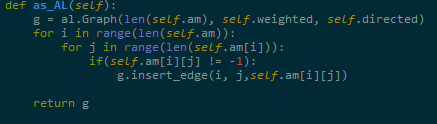
In the adjacency matrix file the first function I implemented is the delete edge function, can done by checking if the graph is directed, if it is, we set the source and destination to the weight. Otherwise, we set the source and destination to the weight, and set the destination and the source to the weight because the matrix is split down the middle so they need to have the same values on both sides. **O(n)**

The next function implemented in the am file is delete edge which can be done in linear time and the only thing done is to set the source and destination to -1 because -1 represents not having an edge. **O(n)**

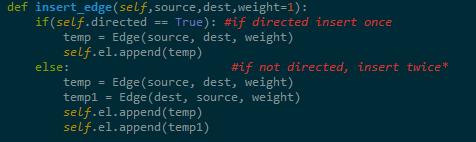
We are also asked to implement the draw function for each representation. The function can be done by setting a variable ‘g’ to an instance of the class al and calling the draw function given to us and calling it with the g variable.

For each of the graph representations we are asked to change the representation of the graph in the file to the other two graph representation. For the adjacency matrix the first one to be done is the as\_el which is the function to represent an adjacency matric to an edge list representation of a graph. The first thing to do is to set a variable ‘g’ to an instance of the edge list representation. The next thing to do is to create a nested loop to get to every element of the list and use the insert edge to input I and j into the graph g. the following piece of code is what would be done for an edge list representation and an adjacency list representation of the graph. **O(n^2)**

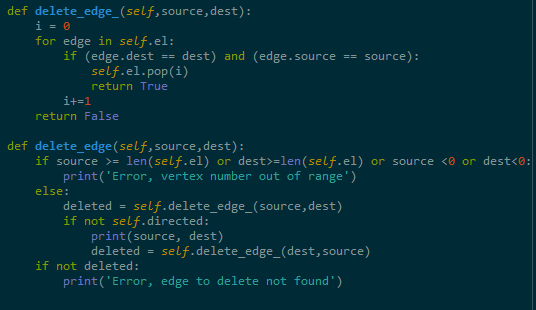




The next thing we are asked to implement are the same functions for an edge list representation of graphs. The first function we are asked to implement is insert edge, for insert edge we check to see if the graph being used is directed. If it is directed we only insert once and if it is not directed we insert twice.



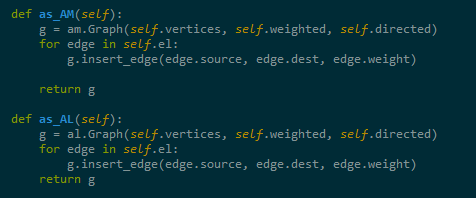
For the delete edge function I used a for loop to iterate through the list and checked if the destination and the source are equal to the node, if it is I used the pop function to remove the node though this function is called in the other delete edge function which checks for length of edge and destination and then creates delete variable to the delete edge returns true or false



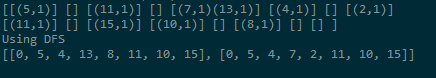
The next function to be implemented was the display which uses a for loop to iterate through the list and print the source destination and the weight. **O(n)**

The draw function is used the same way as the adjacency list where we set the variable g to the al instance and call the draw function with the variable g

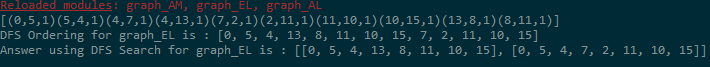
The as am and as al functions are used the same way as the al class but used with the variable set to the am graph and the variable al graph and iterate and through the list and use the insert edge to insert each of the edges in the loop



The following source code is for the part two which implements the graph depth first search and breadth first search to give the solution to the problem being given



The following is the second implementation using an edge list



import graph\_AM as am

import graph\_AL as al

import graph\_EL as el

from queue import Queue

from math import \*

'''

# BFS - use Queue

answer = []

q = []

q.append(0)

while q:

index= q.pop(0)

answer.append(index)

#get all of the possible answer []

for edge in g.al[index]:

if edge.dest not in q:

q.append(edge.dest)

print("Using BFS")

print(answer)

'''

def dfs():

#DFS - use Stack

print("Using DFS")

answer = []

previous = []

rest = []

q = []

q.append(0)

previous.append(0)

split = False

once = False

while q:

index= q.pop()

if once: #save the beginning of the path when you're at state "4"

rest.append(index)

once = False

if len(g.al[index]) == 1 and not split:

for edge in g.al[index]:

if edge.dest not in q:

q.append(edge.dest)

previous.append(edge.dest)

elif len(g.al[index]) == 1 and split:

for edge in g.al[index]:

if edge.dest not in q:

q.append(edge.dest)

rest.append(edge.dest)

elif len(g.al[index]) > 1:

# len = 2

split = True

rest = []

once = True

for edge in g.al[index]:

if edge.dest not in q:

q.append(edge.dest)

else:

# len = 0

answer.append(previous+rest)

rest = []

once = True

print(answer)

def dfs2():

# DFS for graph\_EL using Stack

d = {}

for edge in g.el:

if edge.source not in d.keys():

d[edge.source] = [edge.dest]

else:

d[edge.source].append(edge.dest)

order= []

answer = []

stack = []

prev = []

rest = []

splitted = False

stack.append(0)

while stack:

item = stack.pop()

order.append(item)

if item in d.keys():

if len(d[item]) == 1 and not splitted:

prev.append(item)

for i in d[item]:

stack.append(i)

elif len(d[item]) == 1 and splitted:

if d[item] == 15:

answer.append(prev+rest)

rest = []

else:

rest.append(item)

for i in d[item]:

stack.append(i)

elif len(d[item]) > 1:

splitted = True

prev.append(item)

rest = []

for i in d[item]:

stack.append(i)

else:

answer.append(prev + rest + [15])

rest = []

print("DFS Ordering for graph\_EL is :", order)

print("Answer using DFS Search for graph\_EL is :", answer)

# DFS for graph\_EL using Stack

g = al.Graph(16, directed = True)

g.insert\_edge(0,5)

g.insert\_edge(5,4)

g.insert\_edge(4,7)

g.insert\_edge(4,13)

g.insert\_edge(7,2)

g.insert\_edge(2,11)

g.insert\_edge(11,10)

g.insert\_edge(10,15)

g.insert\_edge(13,8)

g.insert\_edge(8,11)

g.display()

dfs() # change graph to al to use this #AL

#dfs2() #AL